Advanced SHARP APL

Language Elements

Course Introduction

Why you are here:

SHARP APL has evolved since the reference manual was last revised.

Documentation of the changes fragmented, scattered and often unavailable.

Power of SHARP APL often under-exploited or misunderstood.

SHARP APL is your primary tool. Even tiny increases in your grasp pay off.

Fewer and fewer Gurus around.

What you will learn

Material addresses two needs:

It explains those elements of SHARP APL added or changed since the Reference Manual was published.

It describes how the underlying philosophy of SHARP APL has evolved.

Notation and Conventions Used

'{' and '}' are used to delimit APL symbols and code fragments and other special symbols and pseudo-code. Not used for isolated numeric constants or variable names.

Upper case Roman letters are used for APL variable names.

Lower case Roman letters are used for meta-language elements.

{⍺} and {⍵} are used to represent the left and right arguments of a function.

{*f*} and {*g*} are used to represent function arguments of an operator.

{*n*} and {*m*} are used to represent data arguments of an operator.

{←→} is used between two APL expressions that are equivalent.

{←/→} is used between two APL expressions that are not equivalent.

{←~→} is used between two APL expressions that are vaguely equivalent.

Origin 1 (`⎕io←1`) is assumed throughout.

APL Terminology

Mathematical	Linguistic
Data	Noun
Function	Verb
Monadic Operator	Adverb
Dyadic Operator	Conjunction
Scalar	Item
Vector	List
Matrix	Table

APL Data Types

Data Type	Storage Required (in bits/element)
Boolean/Binary	1
Integer	32
Real/Floating-point	64
Complex/Imaginary	128
Character/Literal	8

APL Functions vs APL Operators

	Functions	Operators
Arguments:	Data only	Data and/or functions
Valence:	Ambivalent	Fixed. Monadic or dyadic but never both.
Result:	Data only	Derived function
Precedence:	Lower	Higher
Evaluation:	Right-to-left	Left-to-right
Monadic Syntax:	Prefix	Postfix
Assignable:	Yes	No

APL Primitive Functions

Names and Symbols

Symbol	Symbol Name	Monadic / Dyadic Function Names
⍴	rho	shape / reshape
⋆	star (?)	exponential / power
⌊	?	floor / minimum
⌽	?	reverse / rotate

APL Operators and Derived Functions

Names and Symbols

Symbol	Symbol Name	Derived Function Names
/	slash (?)	compress / replicate / reduce
\	backslash (?)	expand / scan
⍤	paw	with / rank / cut

Simple Arrays

APL has only one data structure - the rectangular array.

The structure of a "simple array" is completely defined by the number and length of each "axis".

The "Shape" of an array is the length of all its axes.

The "Rank" of an array is the number of axes it has.

The "Count" of an array is the number of elements it has.

Simple arrays must be either entirely numeric or entirely character.

Special Simple Arrays

Arrays with a rank of 0 are called "scalars".

Arrays with a count of 1 are called "singletons".

Watch Out: Singletons are often treated as if they were scalars, ***BUT NOT ALWAYS***.

Arrays with a count of 0 are called "empty arrays".

There is no "empty scalar". Why?

Arrays with identical counts are not always identical.

Obviously {`2 3⍴1 ≠/→ 6⍴1`}.

Not so obviously, {`1 ≠/→ ,1`} and {`0⍴1 ≠/→ 1 0⍴1`}.

For arrays to be identical, their shapes must be identical.

Non-Simple Arrays

Early implementations of SHARP APL had only simple arrays.

"Enclosed", "nested", "boxed", "general", and "non-simple" all mean the same thing.

An enclosed array is one in which each element of the array may be "disclosed" (or "opened") to give a further array.

This provides a way to build an "array of arrays", hence the term "nested arrays".

Enclosed arrays are, or can be, displayed with boxes drawn around them.

A simple scalar is not the same as the same scalar enclosed.

```
      {1 ≠/→ <1}.
```

Arrays may be enclosed to any level.

Functions and Operators

Functions that are built-in to the APL language are called "primitive functions".

'Plus' {+}, 'and' {∧}, 'rho' {⍴}, 'take' {↑}.

Functions that result from the use of an operator are called "derived functions".

'Plus-scan' {+⍀}, 'and-dot-equals' {∧.=}, 'two-replicate' {2/}.

A given APL symbols may represent a function, or an operator, ***BUT NEVER BOTH***.

The most important attributes of a function are its "valence" and "rank".

Frames and Cells

Any array may be viewed as a collection, (or "frame"), of similar sized parts, (or "cells").

Consider {X←2 3 4⍴⍳24}. The rank of X is three and that means we can choose between four different ways of describing X. Four different ways of looking at the structure of X.

a) A 3-dimensional array of scalars.

```
2 3 4 ⍴ ⍳
```

b) A 2-dimensional array of vectors.
(i.e. a 2 by 3 table of 4 element lists).

```
2 3 ⍴ 4
```

c) A 1-dimensional array (vector) of matrices.
(i.e a 2 element list of 3 by 4 tables).

```
2 ⍴ 3 4
```

d) A 0-dimensional array of 3-dimensional arrays.

```
⍴ 2 3 4
```

Frames and cells are derived by splitting the shape vector into 2 pieces:

The first part becomes the shape of the frame.

The second part defines the shape of the cells.

frame ⍴ cells

Cells may be defined in either "absolute" or "relative" terms.

Major cells allow any array to be considered as a vector of something.

Rank of Functions

Almost all functions have a rank of 0, 1 or {¯} (meaning "infinite" or 'unbounded").

Functions with a rank of 0 are called "scalar functions".

'Plus' {+}, 'minus' {−}, 'reciprocal {÷}, 'disclose' {>}.

Some monadic functions have a rank of 1.

'Rotate' {⌽}, 'f-reduce' {f/}, 'f-scan' {f\}.

Most other functions have "infinite" or "unbounded" rank.

Group 1 (Nice)

Monadic 'ravel' {,}, Monadic 'transpose' {⍉}.

Group 2 (Nasty)

Monadic 'reverse-down' {⊖},
'f-reduce-down' {f⌿},
'f-scan-down' {f⍀}.

Dyadic Functions with Equal Ranks

Dyadic functions have two ranks - the "left rank" and the "right rank".

Dyadic functions with a left and right rank of 0 are called "scalar functions".

'Plus' {+}, 'times' {×}, 'divide' {÷}, 'power' {⋆}.

Both arguments must the same shape. Their shapes must "conform" or "agree".

Exception: If one of the arguments is a scalar, it is reshaped to match the other argument. This is called "scalar extension".

Dyadic functions with left or right rank greater than zero were called ***"mixed functions"*** in some early APL manuals.

Dyadic Functions with Differing Ranks

'Rotate' {⌽} has a left rank of 0 and a right rank of 1.

```
      {1 0 1⌽3 4⍴⍳12}.
```

```
      {1⌽3 4⍴⍳12 ←→ 1 1 1⌽3 4⍴⍳12}.
```

Scalar extension again.

```
      {(2 3⍴0 1)⌽2 3 4⍴⍳24}
```

but not

```
      {(6⍴0 1)⌽2 3 4⍴⍳24}.
```

Why?

'Membership' {∈} has a left rank of 0 and an infinite right rank.

```
      {'apl'∈'sharp' ←→
```

```
          'apl'∈2 3⍴'sharp' ←→
```

```
              'apl'∈2 3 4⍴'sharp'}
```

The Rank Operator

The rank(s) of a function can be changed using the rank operator.

The rank operator is ***one*** of the uses of the "paw" symbol {⍤}.

Example: Suppose {X← 2 3 4⍴⍳24}
and you want X to be {2 12⍴⍳24}.

{X←,⍤2 X} will ravel each matrix independently.

{,⍤2} is a derived function.

Similarly, {⍉⍤2 X} transposes each of the rank-2 cells of X.

And {⊖⍤2 X} reverses the order of vectors within each of the rank-2 cells of X.

The general syntax of the rank operator is {f⍤n}.

{f} is the function whose rank is being set.

{n} is the value of the rank you want.

Using non-negative value for {n} produces a derived function with a "fixed" or "absolute" rank.

Using a negative value for {n} produces a derived function with a "relative" or "complementary" rank.

In {f⍤¯1 ω}, the rank of {f⍤¯1} will be {¯1+⍴⍴ω}.

Relative rank is most useful with primitives of infinite rank.

Setting the Dyadic Ranks of a Function

If the {*f*} in {*f*⍤*n*} has a dyadic definition, then {*f*⍤*n*} also has a dyadic definition.

Suppose {*A*←1 2 3 4} and {*X*←2 3 4⍴⍳24}.

{*A*+*X*} gives *RANK ERROR*.
{*A*+⍤¯1 *X*} will add *A* to each row of *X*.

Dyadic functions have two ranks, and both can be changed with the rank operator.

When {*n*} in {*f*⍤*n*} is a single number, the derived function has a left and right rank of {*n*}.

When {*n*} is a pair of numbers, the derived function has a left rank of {1↑*n*} and a right rank of {¯1↑*n*}.

Suppose {*B*←10 20} and *X* is still {2 3 4⍴⍳24}.

{*B*+⍤0 2 *X*} will add the rank-0 cells of *B* to the corresponding rank-2 cells of *X*. So it's adding scalars to matrices.

Watch out: You will get still *LENGTH ERROR*s and *RANK ERROR*s if the arguments do not "conform".

Example {*A*+⍤0 2 *X*}. There are four rank-0 cells in *A* and only 2 rank-2 cells in *X* which gives *LENGTH ERROR*.

Rank can be used to simplify primitives with peculiar rules -
"encode" {⊤} and "decode" {⊥} for instance.

Example: {⍺ ⊤⍤1 0 ⍵ ↔ ⍺ ⊤ ⍵}

Example: {⍺ ⊥⍤1 ⍵ ↔ ⍺ ⊥ ⍵}.

Support for Complex Numbers

Most arithmetic primitives and derived functions handle complex numbers.

You may never use them, ***but you have to guard against them***.

Example: {`0j1 ↔ ¯1*÷2`}. Not *DOMAIN ERROR*.

Example: {`1 ↔ ⎕vi '1j1'`}. Not 0.

Complex numbers take twice the store of floating-point numbers.

Support for Enclosed Arrays

All "structural primitives" have been extended to handle enclosed arrays.

'Reshape' {⍴}, 'catenate' {,}, 'take' {↑}, 'drop' {↓}, etc.

But they do not work "inside the enclosure". They manipulate the boxes, not the contents of the boxes.

Most other primitives do NOT work on enclosed arrays, including arithmetic, comparison and boolean functions.

Example: {(<1 2 3)+<4 5 6} gives *DOMAIN ERROR*.

Example: {(<'A') = <'B'} also gives *DOMAIN ERROR*.

You can add and multiply billiard balls, but not eggs.

Exceptions to this sweeping generalization:

There is a replacement for dyadic {=} that DOES work on enclosed arrays.

Dyadic {⍳} and dyadic {∈} DO work on enclosed arrays.

Fill Elements for Enclosed Arrays

Every scalar in an array must be either simple or boxed. Mixtures are not allowed.

Both "overtake" {⍺↑⍵} and "expansion" {⍺\⍵} used to give *DOMAIN ERROR* when {⍵} was non-simple.

Now, the "fill element" for non-simple arrays is {<⍳0}.

Don't forget:

```
{<⍳0} ←→ {<' '} ←/→ {<0} ←/→ {<'  '} ←/→ 0 ←/→ '  '
```

Can lead to problems if you try to disclose such an array.

{>3↑'*just*'⊃'*text*'} gives *DOMAIN ERROR*.

{>3↑1 2 3 4⊃5 6 7 8} does not.

Watch Out:

{1↑0⍴<'*anything*'} ←→ 0 ***NOT*** <''

LCM {∧} and GCD {∨}

The domain of these functions has been extended from boolean to integer.

Dyadic 'and' {∧} has become "LCM" (Lowest Common Multiple).

Dyadic 'or' {∨} has become "GCD" (Greatest Common Divisor).

More importantly, the result of these functions may also be integer.

Upgrade {⍋} and Downgrade {⍒}

"Grade" here means either "Upgrade" or "Downgrade".

The right argument of grade can now be of any rank. Grade has an infinite right rank.

Result is the grade of the major cells of the right argument:

Grade the rows of *X*:

```
{4 1 2 3 ↔ ⍋X←4 2⍴7 2 8 5 9 2 5 7}.
```

Grade with the rank operator:

Grade within each row of *X*:

```
{4 2⍴2 1 2 1 2 1 1 2 ↔ ⍋⍤1 X}.
```

Grade may be used dyadically with character arguments where the left argument is the collating sequence.

For character vectors {⍺} and {⍵}, {⍺⍋⍵ ↔ ⍋⍺⍳⍵}.

Collating sequence is usually a vector, often {⎕AV}.

Matrix collating sequence allow more sophisticated sorts.

Grade with Vector vs Matrix Collating Sequence

```
      X
bad
BAD
Bad
BAG
bag
Bag

      M
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz

      X[M-X;]
BAD
Bad
bad
BAG
Bag
bag

      V
AaBbCcDdEdFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvW
wXxYyZz

      X[V-X;]
BAD
BAG
Bad
Bag
bad
bag
```

When *M* is used as the collating sequence, grade first orders *X* without regard to case, and only then further orders *X* according to the case of the letters.

Upgrade {⍋} and Downgrade {⍒}

The collating sequence need not contain every character in the right argument - it doesn't have to be "complete".

Since {⍺⍋⍵} ↔ {⍋⍺⍳⍵}, characters not present in {⍺} are considered to lie just beyond the end of the sequence.

```
2 4 6 1 3 5 ↔
      'abc'⍋'ZaYbXc' ↔
           'abc'⍋'XaYbZc'
```

This can be avoided by tacking {⎕⍺⍵} to the end of the collating sequence.

Permissive Take {↑} and Drop {↓}

Previously in {⍺↑⍵} and {⍺↓⍵}, {⍴⍺} = {⍴⍴⍵}.

Now, {⍴⍺} ≤ {⍴⍴⍵}.

For take {↑}, {⍺} is extended using {⍴⍵}.

For drop {↓}, {⍺} is padded with zeros.

Most useful with a right argument of rank 2 or higher.

For scalar *S*, matrix *M*:

{*S*↑*M*} will take *S* rows and ALL columns.

{*S*↓*M*} will drop *S* rows and NO columns.

Can cause problems if you make a mistake in your code, since it may fail, but not where you would expect it.

```
{X←4↑V ⋄ ... ⋄ Y←X[1]}
```
gives *RANK ERROR* if V is not a vector.

Rank of the result is still going to be the rank of {⍵}.

```
1 3 ←→ {⍴1 3 3⍴⍳9} ←→ 3.
```

Take {↑} and Drop {↓} with the Rank Operator

One of the most useful applications of the rank operator.

{¯1↑⍤1 ⍵}. Takes the last column of {⍵} where {2≤⍴⍴⍵}.

"Overtake" with the rank operator:

```
      3↑⍤0 ⍳4
1 0 0
2 0 0
3 0 0
4 0 0
```

Take with both ranks set:

```
      ⎕←T←2 4⍴⍳8
1 2 3 4
5 6 7 8

      ¯1 1↓⍤0 1 T
1 2 3
6 7 8
```

Format {⍕} and {⎕fc}

Format { ⍕ } now accepts a character left argument.

The left argument contains "fields".

Two kinds of fields: "active fields" and "decorators".

Active fields contain only the digits 0 through 9.

Decorators are any other characters. They are passed through and appear unchanged in the result.

Example:

```
{'Date: 1991 12 25' ↔ 'Date: 5555 55 55' ⍕3↑⎕ts}
```

Format {⍕} and {⎕fc}

Active fields may be separated by blanks. Other characters will not work:

Example:
```
{'Date: 5555/55/55' ⍕3↑⎕ts}
```

Returns:
```
{'Date:      /19/91Date:      /  /12Date:      /  /25'}
```

Active fields can be terminated with the digit '6':

Example:
```
{'Date: 1991/12/25' ↔ 'Date: 5556/56/56' ⍕3↑⎕ts}
```

Advantages of {⍕} over {⎕fmt} are clarity, speed and formatting the contents of enclosed arrays.

Monadic {⍕} has the "implicit arguments" {⎕pp} and {⎕pw}.

Dyadic {⍕} ignores {⎕pp} and {⎕ps} but obeys {⎕fc}.

{⎕fc} is a 6-item character vector containing the symbols used for decimal points, commas, etc.

With an empty left argument, dyadic {⍕} works like monadic {⍕}.

Match {≡}

Match is like equals with infinite left and right rank.

Two things match if they are identical in EVERY respect.

Paradox: {'' ≡ ⍳0} even though {1↑'' } ←/→ {1↑⍳0}.

Match {≡} is defined on enclosed arrays. Equals {=} is not.

To get an equals-like function that does work on enclosed arrays, use {≡⍤0}.

You can sometimes substitute {∈} for {≡⍤0}:

{*A* ≡⍤0 *S* ←→ *A* ∈ *S*} for scalar *S*.

You can substitute {≡⍤1} for {∧.=} when looking for a vector in a matrix:

{*M*≡⍤1 *V* ←→ *M* ∧.= *V*}.

Monadic Comma-Bar {⍪}

"Comma-bar" {⍪} is sometimes called the "table" function.

Comma-bar is similar to ravel { , }. Both have infinite rank.

Always returns a result of rank 2, a table.

Used to convert a vector to an n-by-1 matrix ("column vector").

Also used to covert a scalar to a 1-by-1 matrix.

Basically {⍪⍵} ravels the major cells of ⍵.

Handy when building boxed structures of mixed type:

```
{(3 2⍴'abcdef')⊃⍪⍳3}
```

rather than

```
{(3 2⍴'abcdef')⊃⍳3}.
```

{⎕*names*⊃⍪⎕*nums*} and {(⎕*pnames* ⍵)⊃⍪⎕*pnc* ⍵} display well.

Dyadic Comma-Bar {⍪}

Dyadic {⍪} is basically { , [1] }, i.e. catenate along the first axis.

For scalar and vector arguments, {⍪} is exactly like { , }.

Result rank of dyadic {⍪} is higher of the two argument ranks:

```
2 ←→ {⍴1⍪1} ←/→ 2 1.
```

```
6 ←→ {⍴1 2 3⍪4 5 6} ←/→ 2 3.
```

Digression Concerning Dyadic { , }

Dyadic { , } has infinite left and right rank.

It may seem to be rank-1, but it's not. It's nasty:

```
3 5 ←→ {⍴1 2 3,3 4⍴⍳12} ←/→ 3 7.
```

If it were really rank-1, the result shape would be 3 7.

When the arguments to { , } differ by 1, the argument of lesser rank is "promoted" by appending a trailing axis of length 1:

```
2 3 6 ←→ {⍴(2 3⍴'a'),2 3 5⍴'b'}.
```

You can defeat this "feature" by fixing the rank to 1:

```
      X
lit
lat

      'sp',X
slit
plat

      'sp',⍤1 X
split
splat
```

Enclose {<} and Disclose {>}

There is no notation for a boxed array constant.

Boxed arrays must be constructed using enclose or link.

"Enclose" (or "box") has infinite rank.

The result of enclose is always an enclosed scalar. Always.

Enclose works well with the rank operator:

```
      A
 1  2  3  4
 5  6  7  8
 9 10 11 12

13 14 15 16
17 18 19 20
21 22 23 25

      ⍴A
2 3 4
      ⍴<A


      ⍴<⍤2 A
2
      ⍴<⍤1 A
2 3
      ⍴<⍤0 A
2 3 4
```

Disclose

"Disclose" (or "open") has a rank of 0.

Disclose has no effect on a simple array. None.

The result of disclose is not always a simple array.

Recently, disclose has been made "permissive". If the individual scalars do not conform after being disclosed, it forces them all to a common rank and shape by padding.

Only disclose has been made permissive, not the frame builder:

{2 4↑⍤0 1 'abcd'}. Gives *DOMAIN ERROR*. But...

{2 4⍴'ab abcd' ↔ >'ab'⊃'abcd'}.

Disclose does the padding.

Enclose and disclose are "inverse functions" or "duals".

Link {⊃}

The link function is ambivalent.

Monadic link is also called "Conditional" or "Weak" enclose.

Behaves exactly like enclose on simple arrays.

Has no effect on boxed arrays.

Used to ensure something is boxed at least once.

Also used to test whether something is boxed:

_ {⍵≡⊃⍵}. Gives 1 if x is boxed, 0 if x is simple.

Dyadic Link {⊃}

Dyadic link {α⊃ω} is defined {(<α),⊃ω}.

Always encloses its left argument but conditionally encloses its right argument.

Joins the two using comma {,}.

Mainly to construct enclosed arrays simply.

Watch out:

Link can easily lead to double enclosures:

{1 2 3⊃4 5⊃6 7 8} vs {(1 2 3⊃4 5 6)⊃6 7 8}.

Watch out:

The comma in the definition implies scalar extension:

2 3 ↔ {⍴1⊃2 2⍴<2}. Since the right argument is already boxed.

Link has infinite left and right rank.

Reducing the rank is often useful:

{⎕*nums*⊃⍤0 1 ⎕*names*} or {⎕*nums*⊃⍤¯1 ⎕*names*}.

Right {⊢} and Left {⊣}

{⊢} is called "right", or "right tack" or "dex" because it "points" to the right.

Monadic {⊢} is an "identity function". It returns its argument unchanged.

Dyadic {⊢} is also returns its right argument, ignoring the left argument entirely.

```
{⍵ ←→ ⊢⍵ ←→ ⍺⊢⍵}.
```

This can be used to embed comments in the middle of an expression:

```
{1+⊢ 'hardly deserves a comment' ⊢1}.
```

Watch out:

The imbedded comment must be a valid APL data object.

Monadic {⊢} can also be used to force a formal result from an assignment:

```
{1 2 3 ←→ ⊢x←1 2 3}
```

Can be used to separate two adjacent numeric constants, as is common with rank and cut operators:

{*Ai*, ⍤1⊢1 2 3}. This is a monadic use of {⊢}. Why?

Left

{⊣} is called "left", or "left tack" or "lev" because it "points" to the left.

Monadic {⊣} is a "null function". It throws away its argument and returns no result.

Note: Iverson's Dictionary of APL has got it wrong when it says otherwise.

The common use of monadic {⊣} is to toss an unwanted result:

```
{⊣⎕ex 'foo'} or {⊣3 ⎕fd 'foo'}.
```

Dyadic {⊣} is the mirror image of dyadic {⊢}.

It returns its left argument, ignoring the right argument:

```
{⍺ ←→ ⍺⊣⍵}.
```

- The main use of dyadic {⊣} is to glue code together:

```
{A←'crazy'⊣B←'glue'}
```

Watch out:

{⊣} is a function whereas {⋄} is "punctuation". The normal right to-left evaluation holds with {⊣}.

Watch out:

You cannot use {⊣} if the right hand "statement" does not return a result:

```
{X←⎕size 1⊣'foo' ⎕stie 1}.
```

This fails as {⎕stie} returns no result for {⊣} to discard.

Nubseive {≠}

A classic APL idiom is the "nub" function which removes duplicate elements from a list:

```
      {'catsrophi' ↔ nub 'catastrophic'}.

      {10 20 ↔ nub 10 10 20 10 20 20 10}.
```

The usual definition of nub is {((ω⍳ω)=⍳⍴ω)/ω}.

With monadic {≠}, the simplified definition becomes {(≠ω)/ω}.

Since {≠} is defined in terms of {⍳}, it works on boxes too:

```
      X←'cat' ⊃ 'dog' ⊃ 'dog' ⊃ 'cat' ⊃ 'toad' ⊃
'cat' ⊃ 'toad' ⊃ 'dog'
      ≠X
1 1 0 0 1 0 0 0

      (≠X)/X
cat dog toad
```

Nubseive {≠}

Nubseive works on higher-rank arguments by flagging the unique major cells:

```
      □←X←>X
cat
dog
dog
cat
toad
cat
toad
dog

      (≠X)≠X
cat
dog
toad
```

Formally, {≠ω} ←→ {(ω⍳ω)=⍳⍴ω←,<⍤¯1 ω}.

{≠} always returns a vector result. Why?

Raze { ↓ }

Monadic { ↓ } is unique in that it is the only primitive that is designed to work primarily on enclosed arrays.

{ ↓ } glues together the disclosed contents of its argument:

```
      7 2 8 5 9 2 5 ←→ {>↓7 2 8 ⊃ 5 9 ⊃ 2 5}.
```

Cleaner and faster than the common work-around solution:

```
      {□fi ↑7 2 8 ⊃ 5 9 ⊃ 2 5}.
```

Another example:

```
      'alistofwords' ←→
          {>↓'a' ⊃ 'list' ⊃ 'of' ⊃ 'words'}.
```

Watch out: The result of raze { ↓ } is always boxed, even if its argument is not.

Epsilon Under-Bar {⍷}

Dyadic {⍷}, also called "in" or "string search", is a kind of infinite rank version of {∈}.

{⍷} searches for its left argument, in its entirety, within its right argument, and returns a boolean array marking places it found a match:

```
1 0 0 0 0 1 0 0 0 0 0 ↔
    {'mat' ⍷ 'mathematics'}.
```

Contrast to {∈} which has a left rank of 0:

```
1 1 1 ↔ {'mat' ∈ 'mathematics'}.
```

{⍷} looks for its left argument as an indivisible unit.

{∈} looks for its left argument one scalar at a time.

Overlapping occurences of the left argument are marked:

```
{(~'   '⍷ω)/ ω}. Deletes redundant blanks
```

Epsilon Under-Bar {⍷}

{⍷} is not limited to a vector arguments - it can search a matrix for another matrix:

```
      □←A←2 2⍴1 0 0 1
1 0
0 1

      □←B←2 5⍴1 0
1 0 1 0 1
0 1 0 1 0

      A⍷B
1 0 1 0 0
0 0 0 0 0
```

In general, {⍷} works on arrays of any rank.

Shape of the result will match the shape of the right argument.

Operators vs Functions

The Fundamental Distinction between a Function and an Operator:

> A Function is applied to data and produces data.
>
> An Operator is applied to data or functions and produces a derived function.

Monadic operators may produce two classes of derived function.

Dyadic operators may produce four classes of derived function.

In all cases, the derived functions may be ambivalent.

Derived functions have ranks, just like primitive functions.

The Domain of Operators

Operators have domains, just like functions.

Presently in SHARP APL, derived and user-defined functions are outside the domain of all operators.

In practice, many operators are defined on only a subset of primitive function.

Terminology Note:

Iverson has taken to calling monadic operators 'adverbs' and dyadic operators 'conjunctions'. Not everyone is comfortable with this terminology yet.

Anomalous Operators and their Syntax

Some operators have a peculiar syntax. Most peculiar.

Axis Operator {`f[n]`}

Applies only to:

The primitive function {`⌽`}, {`⊖`} and {`,`}

The derived functions {`f/`}, {`f⌿`}, {`f\`} and {`f⍀`}

The axis operator is anomalous in many ways:

It has a peculiar syntax involving two symbols {`[`} and {`]`}.

It can be used with certain derived functions, not just primitives.

It uses symbols that are also used for a function, namely 'indexing'. Of course the indexing function itself has a weird syntax.

Outer Product {`∘.f`}

Outer product is anomalous in its use of the jot {`∘`} symbol. Jot is neither a function nor a data value. It is functioning as a kind of place holder.

The "Slash" Operators {n/} & {n≠}

{f} and {g} will stand for functions.

{n} and {m} will stand for data.

{/}, {≠}, {\}, {⍀} are all monadic operators.

All accept either data or function arguments.

All produce monadic derived functions.

Compression and Replication {n/} & {n⌿}

The derived function {n/} is commonly called "compress" or "compress-last".

The derived function {n⌿} is often called "compress-down" or "compress-first".

Previously, the data argument to {/} and {⌿} had to be boolean.

Now it can contain non-negative integers:

{(1+ω=' ')/ω}. Double all spaces in string {ω}.

{3/ω}. Triplicate each column of array {ω}.

{3⌿ω}. Triplicate the major cells of {ω}.

Monadic Functions Derived from the Dot Operator $\{f.g\}$

The derived function $\{f.g\}$, commonly called "inner product", now has a monadic definition as well.

The derived function $\{-.\times\}$, when used monadically is the "determinant" function of matrix algebra:

Other $\{f.g\}$ functions also have monadic definitions.

Composition Operators $\{f\ddot{\circ}g\}$ & $\{f\ddot{\bigcirc}g\}$

WARNING: This is tough stuff because...

The SHARP APL implementation of these operators is incomplete.

There are many restrictions on the domain of these operators.

You must keep many things in your mind at once.

BUT, they are interesting and indispensable for serious work with boxed arrays.

$\{\ddot{\circ}\}$, called "paw", and $\{\ddot{\bigcirc}\}$, called "hoof" are dyadic operators.

When both arguments to either operator are functions, as in $\{f\ddot{\circ}g\}$ and $\{f\ddot{\bigcirc}g\}$, the derived functions created are called "compositions" of the component functions.

This is read in a variety of ways. I prefer just "f with g".

Formal Definition of Composition

```
1:     f∘g ω  ←→ f (g ω)          monadic f, monadic g
2:   α f∘g ω  ←→ (g α) f (g ω)    dyadic f, monadic g
3:     f⍤g ω ←→ f (g ω)           monadic f, monadic g
4:   α f⍤g ω ←→ f (α g ω)         monadic f, dyadic g
```

Note: The derived functions on lines 1 & 3 are exactly the same.

Composition Operators {$f\ddot{\circ}g$} & {$f\ddot{O}g$}

By "composing" {f} and {g}, they become bound together tightly.

Rather than {g} doing something to all of {ω} and returning a result for {f} to do something further, {f} and {g} work jointly on {ω}.

The rank of the derived function {$f\ddot{\circ}g$} is the rank of {g}.

The {g} in {$f\ddot{\circ}g$} is almost always {>}.

Therefore, the rank of {$f\ddot{\circ}g$} is almost always 0.

Composition Operators {f⍤⍤g} & {f⍤g}

```
      □←X←'The'⊃'GST'⊃'is'⊃'loved'⊃'by'⊃'all'
The GST is loved by all

      ⍴⍤⍤>X
3
3
2
5
2
3
```

{⍴⍤⍤>} is a derived function being applied monadically.

The rank of this function is 0 because the rank of {>} is 0.

As each element of *X* is individually disclosed, the shape function {⍴} is applied and the result of {⍴} is saved.

When all six intermediate results have been computed, they are assembled together along a leading axis.

The shape of each individual result is {⍳1}, not {⍳0}, so the shape of the final result is 6 1, not 6.

This is NOT the same as {⍴>*X*}, which returns 6 5.

We can also use the function {⍴⍤⍤>} dyadically.

Composition Operators {f⍤⍤g} & {f⍥⍤g}

```
      X
The GST is loved by all

     4 ⍴⍤⍤>X
TheT
GSTG
isis
love
byby
alla
```

Both the left and right rank of {f⍤⍤g} is 0.

The left argument is extended to match the right (normal scalar extension).

Corresponding pairs are disclosed and the reshape {⍴} function is applied dyadically.

Each individual result is saved.

When everything is done, the six individual results are catenated along a leading axis.

Six vectors of length four gives a result shape of 6 4.

Composition Operators {f⍤⍤g} & {f⍥g}

Watch out:

Each individual result be the same shape, or else the function will signal *DOMAIN ERROR* when it tries to glue them together:

```
{⍴⍤⍤> 1 2 ⊃ 2 2⍴⍳4}
```
. Gives *DOMAIN ERROR*.

The shape of the first partial result is { ,2}

The shape of the second is 2 2.

These do not conform and are not padded.

Inverse Functions

Inverse functions are two monadic functions that, more or less, cancel each other's effect.

Disclose and enclose, when applied in the correct order, are inverse functions:

$\{\omega \leftrightarrow >\!<\omega\}$.

So are log and exponential:

$\{\omega \leftrightarrow \circledast\star\omega\}$.

Many functions are their own inverse:

$\{\omega \leftrightarrow --\omega\}$.

$\{\omega \leftrightarrow \div\div\omega\}$.

$\{\omega \leftrightarrow \phi\phi\omega\}$.

$\{\omega \leftrightarrow \sim\sim\omega\}$.

$\{\omega \leftrightarrow \oslash\oslash\omega\}$.

Formal Definition of Dual Operator $\{f\ddot{\circ}g\}$

```
1:f⍤g ⍵ ←→ g' f g ⍵                    monadic g & g', monadic f
       ←→ g' f⍥g ⍵

2:⍺ f⍤g ⍵ ←→ g' (g ⍺) f (g ⍵)          monadic g & g', dyadic f
         ←→ g' ⍺ f⍥g ⍵
```

Note: g' must exist (g must have an inverse function) or a *DOMAIN ERROR* results.

Note: The derived function $\{f\ddot{\circ}g\}$ is similar to $\{f\ddot{\circ}g\}$.

Uses of Dual

Dual is used to apply functions "inside" enclosed arrays:

```
{<5 7 9 ←→ (<1 2 3) +⍤> <4 5 6}.
```

Read as "plus-dual-disclose" or "plus-under-disclose".

The derived function {+⍤>} is used dyadically.

The rank of {+⍤>} is inherited from {>}, so it is 0.

Both arguments are scalars. In a sense, this is just adding two scalars, even though the scalars are enclosed vectors.

{>} is applied, monadically, to corresponding pairs of left and right arguments, giving 1 2 3 and 4 5 6.

{+} is applied, dyadically, to the intermediate results of {>}, giving 5 7 9.

{<} is applied, monadically, to the intermediate result of {+}, giving {<5 7 9}.

Uses of Dual

```
{11 12 13⊃24 25 26 ↔ (10⊃20) +⍤> 1 2 3⊃4 5 6}
```

Both arguments are vectors. {+⍤>} is rank-0 so it has two separate pairs of arguments.

{>} is applied to {<10} & {<1 2 3}, giving 10 and 1 2 3.

{+} is applied to the result of {>}, giving 11 12 13.

{<} is applied to the result of {+}, giving {<11 12 13}.

The steps are repeated for {<20} & {<4 5 6} giving {<24 25 26}.

The two intermediate results are assembled together.

The final result is {11 12 13⊃24 25 26}.

More Examples of Dual

```
{(<100) +⍤¨> 2 2⍴5⊃10⊃15⊃20}.
```

Here scalar extension is applied to the left argument right away to make it conform to the shape of the right argument.

In effect, this becomes `{(2 2⍴<100) +⍤¨> 2 2⍴5⊃10⊃15⊃20}`.

The rest is as before. The result is `{2 2⍴105⊃110⊃115⊃120}`.

```
{100 +⍤¨> 2 2⍴5⊃10⊃15⊃20}.
```

Another, easier, way to get the same result. Why?

```
{100 +⍤¨> 2 2⍴5 10 15 20}
```

This also gives the same result. Why?

Cut $\{n\ddot{\circ}f\}$

The three ways to use Paw $\{\ddot{\circ}\}$:

Rank Operator $\{f\ddot{\circ}n\}$

Composition Operator $\{f\ddot{\circ}g\}$

Cut Operator $\{n\ddot{\circ}f\}$

Cut provides for the "partitioned" application of $\{f\}$.

The derived function $\{n\ddot{\circ}g\}$ is ambivalent.

When used dyadically, the left argument specifies where to make the cuts in the right argument.

When use monadically, the right argument is cut by taking the first major cell as a delimiter.

$\{n\}$ is one of `¯2 ¯1 0 1` or `2`.

Using Cut

```
      1⍤⍴ ' A string of text'
2
7
3
5
```

The derived function {1⍤⍴} is being used monadically.

Cut points are determined by the first element of the right argument, a blank in this case.

The 1 in {1êÒ} specifies that the cut points mark the beginning of each partition.

{⍴} is applied to each piece and the results are catenated together giving {¯2 7 3 5}.

```
      1⍤< ' A string of text'
+--+ +-------+ +---+ +-----+
| A| | string| | of| | text|
+--+ +-------+ +---+ +-----+
```

The most common use of cut - to box sections of an array.

The variants of Cut

1 means leading cut points, but retain them.

```
     1⁚< 0 7 2 8 0 5 9 0 2 5
+----------+ +--------+ +--------+
| 0 7 2 8 | | 0 5 9 | | 0 2 5 |
+----------+ +--------+ +--------+
```

¯1 means leading cut points, and toss them.

```
     ¯1⁚< 0 7 2 8 0 5 9 0 2 5
+--------+ +------+ +------+
| 7 2 8 | | 5 9 | | 2 5 |
+--------+ +------+ +------+
```

2 means trailing cut points, but retain them.

```
     2⁚< 0 7 2 8 0 5 9 0 2 5
+--------------+ +----------+
| 0 7 2 8 0 5 | | 9 0 2 5 |
+--------------+ +----------+
```

¯2 means trailing cut points, and toss them.

```
     ¯2⁚< 0 7 2 8 0 5 9 0 2 5
+------------+ +--------+
| 0 7 2 8 0 | | 9 0 2 |
+------------+ +--------+
```

0 is left as an exercise for reader.

Dyadic use of Derived Cut-Functions

A boolean left argument may be used to specify the cut points.

{n} behaves as before:

```
      (1 0 0 1 0 1 1 0 0) 1⍤<⍳9
+-----+ +---+ +-+ +-----+
|1 2 3| |4 5| |6| |7 8 9|
+-----+ +---+ +-+ +-----+
```

Monadic cut is really just a special case of the dyadic form.

The result is a series of boxed vectors, in particular, the 6.

Cut with Higher Rank Right Arguments

Cut can partition an array of any rank.

With higher rank arrays, it cuts along the first axis:

```
      X
1 2 3
4 5 6
7 8 9

      (1 0 1) 1⍤<X
+-----+ +-----+
|1 2 3| |7 8 9|
|4 5 6| +-----+
+-----+
```

Note the parentheses surrounding the left argument. Why are these needed?

Why does {f⍤n⊢ω} work but not {α⊢n⍤f ω} or {α⊣n⍤f ω}?

Cut can also be used monadically with higher rank arguments:

```
      X
a
dog
and
a
cat

      1⍤<X
+---+ +---+
|a  | |a  |
|dog| |cat|
|and| +---+
+---+
```

The cut point is the leading major cell, in this case 'a '.

{⎕*fmt*} with Enclosed Arrays

{⎕*fmt*} is very peculiar.

Its right argument is a strange beast sometimes called a "semicolon list".

In this way {⎕*fmt*} can format multiple arrays at once, even if they are of different types and shapes:

```
{⍺ ⎕fmt (1 2 3;'abc')}.
```

Two problems with this:

It does not follow "normal" APL since {;} is not a function.

{(1 2 3;'abc')} can't be assigned since it is not an APL array.

It can be difficult to construct the right argument to {⎕*fmt*} under program control without resorting to {⍎}.

The same effect can now be got by using enclosed arrays:

```
{⍺ ⎕fmt (A;B)} ←→ {⍺ ⎕fmt A⊃B}.
```

Fixing Functions with {⎕fx} and {3 ⎕fd}

{⎕fx} and {3 ⎕fd} are used to set, or "fix", the definition of a user-defined function.

{⎕fx} takes a character matrix right argument.

{3 ⎕fd} takes either a character matrix or a character vector with embedded CR characters between each line.

{3 ⎕fd} used to insist that a vector right argument contain bracketed line-numbers and a leading and trailing {∇}, in other words, the same form as the result of {1 ⎕fd}.

Now it doesn't care:

{3 ⎕fd 1↓,⎕cr, ⎕cr 'foo'} will work.

Easier to build a function under program control.

Fixing Functions with {□fx} and {3 □fd}

It is now possible to fix "invalid" functions.

Both {□fx} and {3 □fd} now let you fix just about any old thing, even if it contains serious errors, including:

Blank lines

Invalid □-names

Mismatched quotes

Illegal characters

Allows importing of code from foreign APL systems.

Watch out: You won't know there are errors until you try to execute the function.

Fixing Functions with {□*fx*} and {3 □*fd*}

You can now fix functions on the SI Stack.

Previously you could not re-fix a function on the SI stack, i.e. an active, pendant or suspended function.

Now, you can pretty much muck around as you please.

Can be a great help when debugging, especially with Logos.

Summary of what can be done:

"Local name" means any name made local to a function.

"Explicit" locals are listed in the function header.

"Implicit" locals are line labels.

Deleting a Local Name:

"Exposes" the next most global value for the name.

Adding a Local Name:

A new local gets the value it would normally have, if any.

Changing a Line in a Function:

OK unless you change a line that is partially executed.

{□create}

You can now specify the starting component for a new file:

```
{'foo' □create 1 42}
```

Creates a file 'foo' such that `42 42 ↔ {2↑□size 1}`.

Simplifies making a copy of an existing file that does not begin at component 1:

```
{'source' □stie 1 ◇ 'target' □create 2,1↑□size 1}.
```

Dyadic {□fhold}

{□fhold} is used to establish cooperative serialization on a file.

Whenever monadic {⍢fhold} is issued, all files currently held are released before the "new" hold takes effect.

This gives another task the opportunity to grab the file.

Dyadic {⍣fhold} provides an alternative.

{1 □fhold ⍵} will hold the files listed in {⍵} without first releasing any files already held.

This is, in effect, a "cumulative" hold.

Dyadic {□fhold} returns the tie numbers of all files currently held.

{1 □fhold ''} can be used to query held files.

Watch out:

There is still no way to tell in advance if a file is held by another task, and there never will be. Why?

Dyadic {□fhold} can be used to "safely" release held files.

{¯1 □fhold ⍵} will release the files listed in {⍵} while continuing to hold any held files not listed in {⍵}.

{¯1 □fhold ⍵} also returns a vector of currently held files.

{□*fhold*} by Range

{□*fhold*} can be used to hold a range of file components.

The argument to {□*fhold*} may be a table with up to four rows:

The 1st row contains the file tie number(s).

The 2nd row contains the file pass number(s).

The 3rd row contains the first component to hold.

The 4th row contains the last component to hold.

Omitting the 4th row will hold a single file component.

Watch out: The documentation implies that you can hold more than one range of components within a single file, but in fact, this is not allowed.

This allows two or more tasks to hold a range of components as long none of the ranges overlap:

Task 1 could hold components 1-10,

Task 2 could hold components 11-20, and

Task 3 could hold components 21-30.

If task 4 tries to hold any of components 1-30, it will wait.

{□rdfi}

{□rdfi} gives information about a file much like the way{□rdci} gives information about a file component.

{□rdfi} takes a tie number, and optionally a pass number.

{□rdfi} returns the following 4 by 2 table:

	account number	timestamp
File creation	.....	.....
Last □stac	.....	.....
***** reserved *****	.....	.....
Last alteration	.....	.....

{□rdfi} timestamps are encoded the same ways as {□rdci}.

The account numbers also display as floating-point. Why?

"Last set of access matrix" is either the last {□stac} or the default setting established by {□create}.

"Last alteration" is the last {□append}, {□appendr}, {□drop}, {□replace}, {□rename} or {□resize}.

The permission code for {□rdfi} is 65536.

{□copy}

After years of painstaking research and development, it is now possible to have the system copy a file for you!

Syntax: {'*file-name*' □*copy tn* [,*pn1* [,*pn2*]]}.

{□*copy*} works much like {□*create*} and {□*rename*}.

The 'file-name' may include an account number and file-size limit.

{*pn1*} is the pass-number used to tie the source file.

{*pn2*} will be explained later.

{□*copy*} duplicates a file in all respects except:

file name

access matrix

file-size (if you specified one)

In particular, the {□*rdci*} for each component IS copied.

The permission code for {□*copy*} is 131072.

Library Access Controls

Every file has a "file access matrix". Always has.

You can now establish a "library access matrix".

Summary of Previously Unavailable Facilities

Create a file in another library, i.e. under another account.

Rename a file to another library, i.e. change the account.

List the files in another library even if you have no other access to them.

Read and set the library access matrix of another library.

Each of these operations has an associated "permission code".

Volume Classes

The file system partitions files into "Volume Classes".

Volume classes are identified by integers.

You can specify a volume class as part of a file name in those {□}-functions that take file names.

For {□*create*}, {□*tie*}, {□*stie*}, {□*rename*}, {□*erase*} and {□*copy*}, the volume class precedes the library number which must be included:

```
{'12 3014912 polygon' □create tie}.
```

Creates file {'3014912 *polygon*'} in volume class 12.

For {□*lib*}, the volume class precedes the library number, and a library pass number must be included:

```
{□lib 12 3014912 0}.
```

List files in library 3014912, volume class 12.

Shared Variables

Shared Variables - General

Shared variables allow two tasks running on the same host to communicate information immediately and directly, and, if done correctly, in a controlled fashion.

Allows the construction of "server tasks", which can coordinate, and provide a central job-handling function.

Not really "shared" variables. That is, it is not shared storage with two pointers; rather, distinct variables which are kept in sync (for the most part) by the APL system.

Sharp APL provides a number of system commands/variables which can be used to establish sharing, control the setting/use of the variables, and detect changes to the "state" of the variable(s).

Actually three entities involved: the two sharing tasks (which we will call "TaskA" and "TaskB"), plus the Shared Variable Processor (the "SVP").

When TaskA "sets" (assigns) a var that it has shared with TaskB, the value is copied into the SVP, and then on to TaskB when TaskB references (uses) the variable. (Of course, the reverse can happen as well: TaskB sets, TaskA uses).

Assume that TaskA and TaskB have established a shared variable between them. When TaskA sets the variable, it is copied to the SVP. When TaskB uses the variable, the SVP passes it to TaskB and deletes its own copy (to save space, etc). Thus, only 2 of the 3 entities "knows" the value of the variable at any given time.

Shared Variables - Processor ID

In order to use shared vars, you must be a uniquely identifiable task on the APL system (so other tasks can point to you).

Account number only partially identifies you, since concurrent tasks on the same account are possible.

Therefore, use a "**clone-id**": an arbitrary integer number which, when coupled with your account number, uniquely identifies you. The (**account number, clone-id**) pair is known as your "**Processor ID**".

Only one task running on an account can have a given clone-id at any particular point in time (ie, clone-id must be unique across tasks running on the same account).

Clone-id defaults to zero in a clear workspace.

Set the clone-id using ⎕*svn*. **Freeze** the clone-id by making a share offer (example assumes no outstanding share offers, and the various clone-ids are available for use):

```
     ⎕svn 888
888
     ⎕svn 1212 ⍝    ok to change clone-id
1212
     (1 2⍴1234567 999) ⎕svo 'z'
1
     ⎕svn 323 ⍝     clone-id is frozen
1212
```

Choosing a clone-id:

Generally set clone-id for one of two reasons:

(a) Prevent clone-id "collision": two tasks on same account using the same clone-id (especially 0, which is the default).

(b) Want to be identifiable (eg, a server that tasks can share with). Requires an "fixed" clone-id (from ⎕*run* to ⎕*run*).

If (a), one convention is to use `''⍴⎕`*runs*. If all tasks on an account use this convention, guaranteed not to have collision.

If (b), choice is arbitrary, but if using convention (a) for other tasks on same account, should avoid numbers in the range `1000-9999`, since this is the domain of {`''⍴⎕`*runs*}. One convention is to choose a number between `1-999`, although any integer will work.

Shared Variables - System Commands/Variables (Part I)

□*svn* ω **Set/query the clone-id.**

¯1 ←→ Query current clone-id
+*n* ←→ (Attempt to) set clone-id to n.

□*svq* ω **Used to detect incoming share offers.**

⍳0 ←→ ProcID's with outstanding share offers.
ProcID ←→ Vars that this Processor wishes to share.

[α] □*svo* ω **Offer vars for sharing, or query on the coupling of vars.**

α ←→ ProcID to share with.
ω ←→ Vector/matrix of variable name(s) (possibly with surrogates).
If α exists, var(s) to share with α.
If no α, query the level of coupling of var(s).

□*svr* ω **Retract share offer of var(s).**

ω ←→ Vec/mat of var name(s).

The initial coupling level of a variable is 0. When TaskA offers to share a variable, the level increases to 1. When TaskB offers to share the same variable with TaskA, the level increases to 2, and the variable is now "fully coupled".

One more way to choose a clone-id: If you have to write code which MUST work on an account, and you have no idea what clone-id choosing convention the workspace you are running in is using, can definitely select a clone-id by looping on □*svn*:

```
→(0<□svn ¯1)⍴GOON ⍝ clone-id already set?
n←1
L1:→(0<□svn n)⍴GOON ⋄ n←n+1 ⋄ →L1
GOON:
```

Shared Variables - A Simple Example

Consider TaskA and TaskB wanting to establish a shared variable. Both tasks have agreed to the variable name. and know the Processor ID's of the other task. Both tasks currently have a clone-id of 0 (which is the default in a clear ws).

TaskA (account 1234567)

```
      □svn 100
100
      V←'Hello'
      (1 2P7777777 200) □svo 'V'
1
```

```
      V
Hello yourself.
      □svr 'V'
2
      V
Hello yourself.
```

TaskB (account 7777777)

```
      □svq ''
      □svn 200
200
      □svq ''
1234567 100
      □svq 1234567 100
V
      (1 2P1234567 100) □svo
'V'
2
      V
Hello
      V←'Hello yourself.'
```

```
      V←'Are you there?'
      □svo 'V'
1
      □svr 'V'
1
      V
Are you there?
```

Shared Variables - System Commands/Variables (Part II)

How do tasks know who last set the variable (ie, should they look at it yet?), and what prevents both tasks from trying to set the variable at the same time?

□svs ω **Shared variable state:**
Who knows the value of the variable(s)?

ω ←→ Vector / matrix of shared variable name(s).

0 0 1 1 ←→ Both you and partner are aware of value.
1 0 1 0 ←→ You have set, partner has not used.
0 1 0 1 ←→ Partner has set, you have not used.

α □svc ω **Shared Variable Control:**
Prevent out of sequence set/use by yourself or by your partner.

ω ←→ Vector / matrix of shared variable name(s).
α ←→ 4 element vector (4 column matrix) boolean control mask:

1 *inhibits*	*set*		*use*	
by	*you*	*partner*	*you*	*partner*
until	*partner has used or set*	*you have used or set*	*partner has set*	*you have set*

Therefore, 0 0 1 0 **□svc** *'shvar'* prevents you from using *shvar* until your partner has set it's value. If you attempt to use it before your partner sets it, you will "hang" (without a timeout) until it gets set, the share breaks, or the break key is hit (if you are a t-task).

Watch out: Unless both tasks agree upon the **□svc** protocol being used, the result will almost surely be one of:

Lost data (two sets without an intervening use),
Repeated data (two uses without an intervening set), or
A hung task (waiting for a set/use that will never come).

□svc - Some Common Settings

0 0 0 0	No interlock. Tasks are in danger of duplicate sets/reads if no other form of control is used.
1 1 0 0	Neither partner can set a new value until the other has used it (sometimes called half-duplex).
0 0 1 1	Neither partner can use the variable until the other has set it (another form of half-duplex).
1 0 0 1	You can't set until partner has used, and your partner can't use until you have set (sometimes called simplex transmission from you to your partner).
0 1 1 0	You can't use until partner has set, and your partner can't set until you have used (sometimes called simplex transmission from your partner to you).
1 1 1 1	Full interlock: reversing half-duplex. One piece of information moves from TaskA to TaskB, then one piece of info moves from TaskB to TaskA, etc.

□sc - State Change Variable

When dealing with shared variables, often want to know when your "state" (vis-a-vis shared variables) has changed (Any new offers? Partner retracted an offer? New shared variables sets/uses?).

Generally want to wait (ie, do nothing) until the state changes.

Can use □dl loop to do this, but is imprecise (eg, you perform a {□dl 60}, someone offers to share one second later, they wait a full minute before the couple can take place).

System variable □sc will freeze a task (like □dl), but will return immediately whenever your shared variable state changes. A "change" includes:

1. **a new offer**
2. **retraction of an existing offer** (whether coupled or not)
3. **set of a shared variable**
4. **use of "freshly" set shared variable**

(3 and 4 can be stated as "a change to the □svs of a shared variable")

Any of 1-4 above is known as a "**post**", as in "□sc is posted whenever a new share offer is received".

Result of □sc is always 1, unless the clone-id is improperly set; then result is 0. Thus, result is usually ignored (eg, {⊣□sc}).

□sc can be assigned a value of the same domain as the argument to □dl. Essentially, assigning this value causes a post to occur at a point in the future, waking you up from your □sc sleep. For example:

```
□sc←30
⊣□dl 10 ⍝          wakes up in 10 seconds
⊣□sc ⍝             wakes up 20 seconds later
```

Note that assigning □sc sets a **maximum** limit before the task will wake up; the task could wake up almost immediately if a change to its state occurs.

□sc for Non-Shared Variable Servers

Most shared variable servers are programmed using a □sc-driven loop.

Servers which process, say, file information, and are not specifically interested in shared vars, may us □sc to allow for an immediate processing of data:

The server sits in a □sc loop. The function which adds / changes data in the file, shares (and then retracts) a junk variable with the server immediately after changing the file. The server wakes up, processes the data, then goes back into □sc sleep.

Such servers will generally work with a □sc timeout, in case a file change occurs, but the changing task fails to post the server.

Note that the retract is a post as well, and that the server will wake up twice - can't be helped.

General Server - Introduction

Model of a general shared variable server which does the following:

Runs as an N-task.

Couples with offers to it from any Processor ID, any variable name.

Once coupled, tasks can send APL commands to the server (in the form of character vectors) to be executed in the local server environment.

Can also send APL packages, which the server will ⎕*pdef* into its local environment.

Really just the "skeleton" for a more elaborate server.

Main function is ***serve***, which is a ⎕*sc*-driven loop. Every time through the loop, couples with new shares, uncouples old (broken) shares, and performs any work requested by existing shares.

General Server - Function: *serve*

```
    ∇ serve cid;Procids;Shutdown
[1]  ⍝ ntask server which processes requests from other tasks
[2]  ⍝ <cid>: integer scalar clone id for the server
[3]  ⍝ global var <Procids>:  n×3 int mat of accts, clone ids
[4]  ⍝            shvar nums of tasks we are sharing with.
[5]  ⍝ subfns: retract, requite, respond
[6]   Procids← 0 3 ⍴Shutdown←0 ⍝  shutdown flag
[7]   →(cid=⎕svn cid)↓ERR ⍝       able to establish cloneid?
[8]   ⊣⎕svo 'SERVERISUP' ⍝        indicate that server is up
[9]  TOP:retract ⍝                retract old shares
[10]  requite ⍝                   requite new incoming shares
[11]  respond ⍝                   respond to sets of shares
[12]  →Shutdown⍴END ⍝             asked to shut down?
[13]  ⎕sc←300 ⋄ ⊣⎕sc ⍝            make sure we wake up
[14]  →(2 ⎕ws 3)[2]⍴END ⍝         APL shutting down?
[15]  →TOP
[16] ⍝
[17] ERR:'clone id not established' ⎕signal 555
[18] END:
    ∇
```

Notes:

Line [7]: Since only one task on an account can have a given clone-id, this line "locks out" other copies of the same server. Can use this fact in other ways as well, eg, ensuring that a group of tasks do their work one at a time (Who needs ⎕*hold*?).

Line [8]: Left argument to ⎕*svo* of 0 indicates "global" share. Any task can couple with this offer, although it doesn't show up in anybody's ⎕*svq* ⍳0. (Why should it not?)

Lines [9 10 11]: The real work. Perform *retact* before *requite* to free up shares before trying to couple more (a task can only have 100 shares at any given time).

Line [13]: Not strictly necessary to set a ⎕*sc* timeout; just to show you how it's done. The timeout is more important if the server has other, non-share related work (eg, monitoring a file for new data, etc).

Line [14]: This system variable is normally 0, but gets set to 1 about 5 mins before APL is shut down. Handy for "end of week" cleanups, etc.

General Server - Function: *requite*

Function *requite*: Couple with new incoming shares.

```
    ∇ requite;ind;id;newprocids;surrogate;i;n
[1]   ⍝ detects offers by new partners, and couples shares with them
[2]    i←0 ◇ n←''⍴⍴newprocids←⎕svq⍳0 ⍝ any new processor ids?
[3]   TOP:→(n<i←i+1)⍴OUT
[4]    id←newprocids[,i;] ⍝ acct/cloneid of next task
[5]    →(0∈⍴surrogate←⎕svq,id)⍴TOP ⍝ has offer been retracted?
[6]    surrogate←surrogate[1;] ⍝ pick first share name offered
[7]    ind←((⍳''⍴⍴Procids)∈0,Procids[;3])⍳0 ⍝ next available num
[8]    ⊣id ⎕svo('V',,'g<999>' ⎕fmt ind),' ',surrogate ⍝ sv "Vnnn"
[9]    Procids←Procids⍪id,ind ⍝ save acct/clonid/shvarnum
[10]   →TOP
[11]  OUT:
    ∇
```

Notes:

There is no guarantee that every user will offer to share a unique variable name with the server, and that none of these names will conflict with existing object names in the workspace. Indeed, if a function to couple with the server is distributed to different tasks, most of the names offered will be the same. Fortunately, the SVP lets us use "surrogate" variable names. Essentially, the server and the partner use different names to reference the same variable.

In this example, we generate local names based upon the "next available integer" algorithm (often used for file tie numbers). The next available number is formatted into a variable name of the form "*Vnnn*" (eg, variable number 12 would be "*V012*").

The Processor ID and variable number of each partner is saved in the matrix <Procids>.

General Server - Function: *respond*

Function *respond*: Perform work requested by existing shares.

```
    ∇ respond;chgidx;nms;nm;z
[1]  ⍝ respond to new share sets by partners
[2]  ⍝ subfn: <process> (written by user)
[3]  ⍝ first, define vector of changed (set) variables:
[4]   z←⎕svs nms←'V','g<999>' ⎕fmt Procids[;3]
[5]   chgidx←z[;2]/⍳''⍴⍴Procids
[6]  TOP:→(0∊⍴chgidx)⍴OUT ⍝  any changed?
[7]   ⍎nm,'←process ',nm←nms[''⍴chgidx;]
[8]   chgidx←1↓chgidx ◇ →TOP
[9]  OUT:
    ∇

    ∇ r←process r
[1]  ⍝  example of a process function
[2]   →(2=⍴⍴⎕pnames r)⍴EXE ⍝ package?
[3]   →('⍎'∈1↑,r)↓L1 ◇ r←1↓,r ⍝ wants to execute something?
[4]  EXE:r←exec r ◇ →END ⍝ call the <exec> fn
[5]  L1:r←⍎r ⍝ all other processing here
[6]  END:
    ∇

    ∇ r←exec ⍵;⎕er;⎕trap
[1]   ⎕trap←'∇ 8 c →RER ∇ 0 1000 c →OER'
[2]   →(2=⍴⍴⎕pnames ⍵)↓XEQ ⍝ package of goodies to use?
[3]   r←⎕pdef ⍵ ◇ →0
[4]  XEQ:r←⍎⍵ ◇ →0
[5]  RER:r←'(no result)' ◇ →0
[6]  OER:r←'*** error:'⊃⎕er
    ∇
```

Lines [4 5]: Test for this function is new sets by partner. Note the simple check of col 2 of ⎕svs result for all shares. Only the "partner knows but I don't" state has a 1 in col 2.

Line [7]: This one line gets the value from the shared var, sends it to the fn <process>, then assigns the result back to the shared var. This will work only if <process> ALWAYS returns a result.

process: Here, just a cover for ***exec***. Other work could be placed from line [5] on.

exec: Note that all errors are trapped (Why?). In particular, ***result error*** (8) is not really an error, and just forces the return of something. Note that we can still cause the server trouble if we pass certain executable strings (Examples?).

General Server - Function: *retract*

Function *retract*: Clean up shares broken by partners.

```
    ∇ retract;ok;nms
[1]  ⍝ retract shares with partners who have
[2]  ⍝ retracted their variable
[3]   →(∧/ok←2=⎕svo nms←'V',⍕g<999>' ⎕fmt Procids[;3])⍴END
[4]   ⊣⎕ex(~ok)⌿nms ⍝ retracts share and erases name
[5]   Procids←ok⌿Procids ⍝ keep those still fully coupled
[6]  END:.
    ∇
```

Notes:

Line [2]: Function driven by checking all shared vars (ie, all "*Vnnn*" vars) to see if any of them have a coupling of less than 2.

Line [3]: Erasing the variable automatically performs a ⎕*svr*. Since we want to erase the var, we would waste CPU if we ⎕*svr*'ed then ⎕*ex*'ed.

General Partner - Introduction

A group of fns to start, communicate with, and stop a general server.

Function *startserver*: Start a server on the current account.

```
    ∇ r←startserver cloneid;⎕sp;z
[1]   ⍝ start a server on this account with cloneid <cloneid>
[2]    r←0 ⋄ →(isup(''⍴⎕ai),cloneid←''⍴cloneid)↓OK
[3]    '*** server is already up' ⋄ →END
[4]   OK:⎕sp←cloneid
[5]    z←⎕run ': genserver servercr 0 ',⍕9000000+cloneid
[6]    →(r←0=''⍴z)⍴END
[7]    '*** unable to start server; ⎕run result: ',⍕z
[8]   END:
    ∇
```

Notes:

Line [2]: Use function *isup* to see if this server is already running; if so, quit.

Line [5]: The number of seconds timeout is set to `9e6+cloneid`. This allows a quick check to see if server is running by using ⎕*runs*.

Function *isup*: Check to see if a server is up and running.

```
    ∇ r←isup server;z
[1]   ⍝ check to see if <server> is up.
[2]   ⍝ <server> is the acct/cloneid of the target server
[3]    ⍎⎕svn ''⍴⎕runs ⋄ →(0∈⍴z←⎕svq server)⍴r←0
[4]    r←∨/(10↑⍕⍤1 z)∧.='SERVERISUP' ⍝ check for magic var name
    ∇
```

Notes:

Line [3]: We use the `''⍴⎕runs` convention to set ⎕*svn*. Note that we really should check to see if ⎕*svn* is already set via `~0∈⎕svn ¯1`.

Line [4]: Look for the global share variable which is offered by the function ***serve*** in the general server. Note that we must **never** couple with this share (Why not?).

General Partner - Communication

Function *talk2*: Establish a share with the server.

```
    ∇ r←talk2 server;i
[1]  ⍝ couple variable 'Svar' with <server>.
[2]  ⍝ <server> is acct/cloneid of the target server
[3]   ⊣⎕svn ''⍴⎕runs ◇ r←i←0
[4]   →(isup server)⍴OK ⍝ is the server running?
[5]   '*** server is not up' ◇ →END
[6]  OK:→(2∈(1 2 ⍴server) ⎕svo 'Svar')⍴SVC ⍝ offer; coupled?
[7]  DL:⊣⎕dl 1 ◇ →(2∈⎕svo 'Svar')⍴SVC
          ◇ →(3≥i←i+1)⍴DL ⍝ wait; coupled?
[8]   '*** server will not couple' ◇ →END
[9]  SVC:⊣ 0 0 1 0 ⎕svc 'Svar' ⍝ don't use until partner sets
[10]  r←1
[11] END:
    ∇
```

Notes:

Line [7]: Wait for the server to answer the couple. Exact timeout depends on whether or not the server is expected to be able to couple quickly. If it is likely that it is performing long tasks, may have to wait longer, or try again.

Line [9]: Perform a ⎕svc to inhibit ourselves from using var too quickly.

Function *sr*: Send request / Receive result:

```
    ∇ r←sr data
[1]  ⍝ send <data> to server; receive result back
[2]  ⍝ can optionally perform <2=⎕svo 'Svar'> at top of fn
[3]   Svar←'▲',,data ◇ r←Svar ⍝     it's that easy
    ∇
```

Notes:

The ⎕svc on line [9] of *talk2* allows us to just set and then use the variable. We will hang on the *r←Svar* until the server returns us a value.

General Partner - Stopping the Server.

Function *stopserver*: Stop a running server.

```
    ∇ r←stopserver cloneid
[1]  ⍝ shutdown a server (cloneid <cloneid>) on this account
[2]   r←0 ◇ →(isup(''⍴⎕ai),cloneid)⍴OK
[3]   '*** server is not up' ◇ →END
[4]  OK:→(2=⎕svo 'Svar')⍴L1
      ◇ →(talk2(''⍴⎕ai),cloneid)⍴L1 ◇ →END
[5]  L1:⊣sr 'Shutdown←1'
[6]   '*** server asked to shut down' ◇ r←1
[7]  END:
    ∇
```

Notes:

Have to ensure that the server is up, and that we are talking to it (lines [2] and [4]). Then it is simply a matter of setting the *Shutdown* boolean in the server task to a 1, and the server will shut down immediately.

The "Infinite Post"

One danger which can arise if one □sc-driven task attempts to use the services of another □sc-driven task is that of the "infinite post" (author's term). Consider the following:

> TaskA and TaskB are two tasks of the General Server type. TaskA, every time around its loop, sets a variable for TaskB to read (could be sending status information to TaskB). Consider the sequence of events:
>
> > TaskA sets the variable
> > TaskB is posted, and wakes from it Ìsc sleep
> > TaskB sees that TaskA's var has been set, and uses it
> > This use is a change of TaskA's state, so TaskA wakes up
> > Since TaskA sets the var every time through it's loop...

Basically, the result is that the two tasks "thrash", TaskA continually setting, TaskB continually using, and each task posting the other non-stop.

Solution? In most cases, TaskA doesn't really need to set the variable EVERY time through the loop. An external mechanism can be used to prevent the constant setting (set a maximum of once a minute, etc).

General Server - Limitations

The general server is a good model, but has limitations:

No security: Anyone can couple, using any variable name. Furthermore, anyone (once connected) can see everything in the server, and even shut the server down. Connection could be restricted to certain accounts, a single (or small group) of variable names, etc.

No protection against ws-full: If a group of people each share a large variable with it, the server will ws-full. Some ws-full protection can be provided by setting a `⎕trap` in the `process` function for ws-full, creating a large temp var (eg, `z←50000⍴'x'`), and having the `⎕trap` expunge the share.

No protection against "dangerous requests" (inadvertant or otherwise): These include passing a naked branch to the server; a `⎕/⍞/⎕arbin`; a `⎕dl`; etc. While it is hard to think of all such strings, some of the more basic ones can be scanned for and disallowed.

Only one "class" of user: Many servers have two classes of user: the basic user (which is the primary use of the server), and a privileged user (which have access to the `exec` fn). This can be accomplished in various ways: privileged users use offer a different share name; or only certain accounts are deemed privileged; different shared variable content or structure; combination of the above, etc.

No crash protection: Since the server is (presumably) serving many tasks, it is important that it try to keep going at any cost. One overall `⎕trap` algorithm would involve discovering who you were serving when the crash occured, and shutting that user out - at least you could serve the rest of the people!

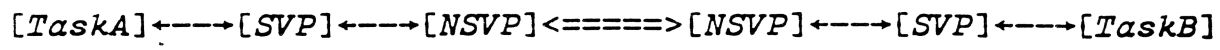
NSVP
The Network Shared Variable Processor

Network Shared Variable Processor -- NSVP
Overview

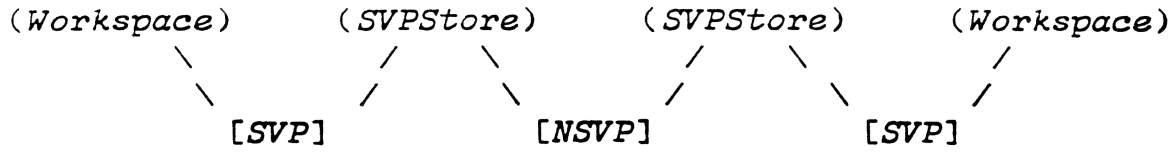
NSVP is an auxiliary processor that, used in pairs, enables programs on different APL systems to communicate using an extended Shared Variable protocol.

Information is transferred over an appropriate communications channel between the two NSVPs, and between the user program and its local NSVP via the local SVP.

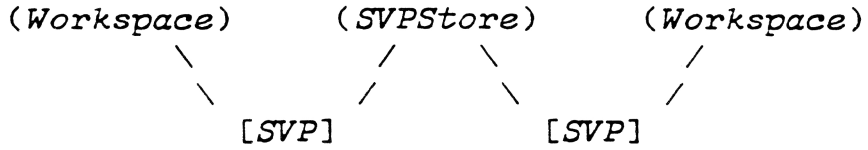
Viewed in terms of processes:



Viewed in terms of storage points and the processes that move data between them:



Contrast this with the more familiar SVP-only situation:



Depending on the requirement and the communications channel, TaskA and TaskB may reside on the same host computer, or on different continents, the only difference being the data transfer time.

The NSVP was evolved based on SVP concepts and is, as far as possible, consistent with the SVP protocol.

Watch Out:

There are differences between NSVP and SVP communications, they are significant, and should be carefully considered when designing systems that will use NSVP. You may not be able, in the general case, to run an existing SVP-based system over NSVP without some alteration or redesign. These differences will be discussed below.

Extensions to Shared Variable Functions
Extensions to □*svn*

Remote systems connected by NSVP are identified both by a character vector system name, and by an integer system id.

Monadic □*svn* has been extended to accept a character vector right argument. If the character vector argument is the name of a system accessible by NSVP (ie, the NSVP link to that system is currently active), the result is the integer id number for that system. The result is ¯1 if your task has no valid clone id. If the character vector argument is not the name of a currently accessible remote system, error 73, "no shares," is signaled.

```
      □svn 'ipsa'
303010

      □svn 'fubar'
no shares
      □svn 'fubar'
      ^
```

Watch Out:

The system id number for any given system may differ from one remote system to another, as it is a local system configuration parameter. Further, system id numbers could change. ***Always*** use □*svn* to establish the system id number of your target system rather than encoding that number in your program as a constant.

Note: □*svn* may be changed at some point to signal error 77, *identification in use*, rather than returning ¯1 in the event that the task does not have a valid clone id, as the current behaviour is inconsistent and may cause problems with □*svo* (see following).

Extensions to Shared Variable Functions
Extensions to ⎕*svo*

Dyadic ⎕*svo* has been extended to accept a three column matrix as its left argument, where previously it only accepted a singleton, two element vector or two column matrix.

The three columns in the left argument are:

System Id, Processor Id, Clone Id

where these identify the task with which you are offering to share those variables named in the right argument. The result continues to be the same, ie, the degree of coupling for the specified variable(s).

The new parameter, **System Id**, must be an integer system number as received from ⎕*svn* (above) or ¯1, the latter specifying the system where your task is running.

Examples:

```
      sysid←⎕svn 'aplc'
      (1 3⍴sysid, 1234567 999) ⎕svo 'ctl'
2
      (1 3⍴¯1 1234567 0) ⎕svo 'ctl'  ↔  (1 2⍴1234567 0) ⎕svo 'ctl'
```

Watch Out:

Since ⎕*svn* can return ¯1, meaning no valid clone id, and ¯1 is a valid system id in this context, explicit tests are advised.

Extensions to Shared Variable Functions
Extensions to □*svq*

Monadic □*svq* has been extended to accept an argument of either <⍳0 or a three-element numeric vector. Previously, □*svq* only accepted an empty vector, a scalar, or a one- or two-element numeric vector as its argument.

□*svq* <⍳0 returns a three-column numeric matrix:

System Id, Processor Id, Clone Id

with one row for each task offering to share with you.

If the argument to □*svq* is a three element numeric vector containing the System Id, Processor Id, and Clone Id, the result is a character matrix containing the variable(s) that the specified task is offering to share with your task.

Examples:

```
      □svq <⍳0
303010 1234567   99
    ¯1     123 4017

      □svq 303010 1234567 99
ctl
dat
```

NOTE: The ¯1 in the result of □*svq* <⍳0 indicates that the corresponding offers originate from a task on the same system as your task.

Functional Limitations

Access Control (□*svc*) Limitations:

Because your task is, in reality, sharing locally with your local NSVP, you can observe and control access only with your local NSVP. Your partner's access level is not reported by □*svc*, which NSVP always sets to `0 1 0 1`. Similarly, you cannot set the access level between yourself and your partner, only between yourself and the local NSVP. The effect of this is that you can only see and alter your own (local) access control.

State Inquiry (□*svs*) Limitations:

The result of □*svs* reflects only the state of the share with respect to the local NSVP. Thus state `0 0 1 1` which would normally indicate that your partner had used the current value in the variable, now indicates that the local NSVP has used the current value.

NSVP will not use the current value until your partner has used the previous value, so in a typical application, state `0 0 1 1` does indicate that it is appropriate to set a new value.

Once the shared variable has been set for the last time, you may safely retract the variable, as NSVP will deliver the final value to your partner before acting on the retraction.

Functional Limitations

Size Limitations:

Any value you set must pass through two SVPs and two NSVPs prior to reaching your partner. Depending on how these processors are configured, a given value may be too large to pass thru one or more of these processors. A value too large to pass thru the Local SVP will result in error 74, "interface capacity exceeded." A value which exceeds the buffer size of either NSVP or the remote SVP cannot result in an error to the sender, and is therefore signaled to the receiver, upon use, as error 6, "*value error*."

Data Limitations:

Because of possible different internal representations of functions among different SHARP APL implementations and versions, NSVP will not pass user defined functions in package data. Again, because the architecture prohibits signalling an error to the sender, error 6, "*value error*," is signaled to the receiver upon use.

Watch Out:

State (0 1 0 1) indicates that the shared variable contains a value and may safely be read. However, a value error on a reference to that variable may result if the data was too large for some processor in the pipeline, contained illegal data (eg, a package with a function) or contained corrupted data. Depending on the nature of the application, it may be necessary to establish a □*trap* for value error.

Functional Differences
Order of Set

As an Auxiliary Processor, the local NSVP cannot necessarily determine in what order two variables are set. Because of this, NSVP cannot guarantee to preserve order of set of multiple variables.

Use of multiple shares between a pair of tasks is most common with auxiliary processors, for example, TSIO. Most commonly in these cases variables 'ctl' and 'dat' are shared in that order, then set in the order

```
dat ← data
ctl ← command
```

For this reason, the convention used by NSVP, where order of set cannot be determined, is to set variables in the reverse of the order in which the share was established, provided that neither variable is blocked (ie, both have been used by your partner).

Watch Out:

NSVP will not delay a set in order to preserve order. Thus, in the ctl/dat example, if dat is blocked, NSVP will proceed to set ctl.

Order of use is not preserved, though it's not clear that this has any practical impact on typical applications beyond occasional anomalous settings of `□svs`.

Functional Differences
Overset

Assuming an appropriate setting of □*svc*, a shared variable may be set multiple times prior to being used by your partner. This is termed "oversetting." In the normal situation with the SVP, your partner will always see the last value set. Thus the SVP is said to have a "***window size of 1***" in communications jargon.

When using NSVP, and assuming no access control is set by the user task, you may again set the shared variable multiple times. However, in the first value will be passed to the remote NSVP, but subsequent values are not read by the local NSVP until the remote NSVP indicates that your partner has read the first value. Thus, after a series of sets by you, your parter will obtain both the First and the Last value set. Thus, NSVP is said to have a "***window size of 2***."

Example:

Assuming variable '*sv*' has been shared via NSVP without setting any access control:

You	*Partner*
sv ← 1	
sv ← 2	
sv ← 3	
sv ← 4	
sv ← 5	
	sv
	1
	sv
	5

Note that setting access control would inhibit the set "*sv*←2" only until the local NSVP had read the value 1. Subsequent sets would be inhibited until the local NSVP had read the value 2, which it would only do after receiving an indication from the remote NSVP that your partner had read the first value. Hence, even with access control set, two values may be "in the pipe" at one time.

Exercises

1. Write a function, *pcb*, that behaves like dyadic comma-bar {⍪} but forces conformance by padding trailing axes. Have "permissive disclose" do the padding if you can.

2. Write a function, *pc*, that behaves like catenate (dyadic comma) {,} but forces conformance by padding. (This is a one-liner if you have the function *pcb* from exercise 1.)

3. Write a function, *pd*, that models "permissive disclose".

4. Write a function, *r←α from ω*, that returns the major cells α, from the array ω.

5. Write as many expressions as you can think of that will add each scalar of vector *V* to the corresponding row of matrix *M* without using rho {⍴}. Assume {(⍴*V*)=1↑⍴*M*}.

6. Write as many expressions as you can think of that will add a vector *V* to each row of matrix *M* without using rho {⍴}. Assume {(⍴*V*)=¯1↑⍴*M*}.

7. Given a vector of boxed words, reconstruct the original sentences. Assume that the last word in each sentence with a period.

8. Find all three letter words of the form (Consonant, Vowel, Consonant) in a vector, eg:

```
CVC 'the cat in the hat'
```

Exercises

9. Given:

```
A←1 2 3
B←(<1),(<2),<3
C←(<<1),(<<2),<<3
```

Fill in the following table:

	⍵←A	⍵←B
<⍤0 ⍵		
⊃⍤0 ⍵		
<⍤> ⍵		
<⍤⍤> ⍵		
⊢⍤⍤> ⍵		
⊃⍤> ⍵		
⊃⍤⍤> ⍵		

10. Write a function, `r←⍺ cddf ⍵`, to model compress dual disclose on the first axis, eg, `r←⍺≠⍤⍤>⍵`.

11. Write a function, `r←⍺ cddl ⍵`, to model compress dual disclose on the last axis, eg, `r←⍺/⍤⍤>⍵`.

Answers to Exercise 1

```
     ∇ r←α pcb ω;lr;ls;rr;rs
[1]   ⍝ model Permissive Comma-Bar
[2]   ⍝ Steve Chapman, April 91
[3]    lr←⍴ls←⍴α⍪rr←⍴rs←⍴ω ⍝ left and right ranks and shapes
[4]    →(0∊lr,rr)⍴l1 ⍝ take fast path if either arg is scalar
[5]    →((1↓ls)≡1↓rs)⍴l1 ⍝ .. or if no padding is required
[6]    ⎕signal(1<|lr-rr)⍴4 ⍝ Domain Error if ranks differ by > 1
[7]    r←(<⍤(1⌈lr-1) α),<⍤(1⌈rr-1) ω ⍝ enclose cells of α & ω
[8]    →0⍪r←>r ⍝ permissive disclose will pad as required
[9]   l1:→0⍪r←α,ω ⍝ fast path for cases req no padding or promotion
     ∇
```

```
     ∇ z←α pcbΔjht ω;m
[1]    m←0⌈¯1+(⍴⍴α)⌈(⍴⍴ω)
[2]    z←>(<⍤m⊢α),<⍤m⊢ω
     ∇
```

```
     ∇ r←top pcbΔrbk bottom;t;⎕ec
[1]    t←(¯1↑1,⍴top)⌈¯1↑1,⍴bottom
[2]    r←(((¯1↓⍴top),t)↑top),((¯1↓⍴bottom),t)↑bottom
     ∇
```

Answers to Exercise 2

```
     ∇ r←⍺ pc ⍵
[1]    ⍝ Permissive Comma
[2]    r←⍺(⍺⍺) pcb⍺⍵ ⍝ uses Permissive Comma-Bar
     ∇
```

Answers to Exercise 3

```
    ∇ ω←pd ω;r;s;mr;⎕ec
[1]   ⍝ model Permissive Disclose
[2]   ⍝ promotes rank and extends shape of cells of ω as required
[3]    mr←⌈/,r←,⍤2⍴⍤⍤>s←⍴⍤⍤>ω ⍝ s←shapes, r←ranks, mr←maximum rank
[4]    s←((mr-r)⍴⍤⍤>1),⍤⍤>s ⍝ extend shapes with 1s on left to max rank
[5]    ω←s⍴⍤⍤>ω ⍝ reshape cells in ω to equal rank
[6]    s←<⌈⌿>,s ⍝ max shape vector
[7]    ω←s↑⍤⍤>ω ⍝ extend cells in ω by overtake on each axis
[8]    ω←>⍤⍤0 ω ⍝ disclose ω (rank 0 to defeat "permissiveness" of >)
    ∇

    ∇ z←pdΔjht ω;m;r;s
[1]    s←⍴⍤⍤>ω ⍝ shapes of the cells of ω
[2]    m←⌈/r←,⍴⍤⍤>s ⍝ ranks of cells and maximum rank
[3]    s←(⌽r∘.<⍳m)+(-m)↑⍤⍤>s ⍝ pad ranks with 1's in leading axes
[4]    z←(<⌈⌿s)↑⍤⍤>(<⍤⍤1 s)⍴⍤⍤>ω ⍝ fix ranks and pad
[5]    z←>⍤⍤0 z
    ∇
```

Answers to Exercise 4

```
     ∇ z←α from ω
[1]   ⍝ returns major cells α from array ω
[2]    z←⊃(<⍤¯1 ω)[α]
     ∇
```

```
     ∇ Δ←α from2 ω
[1]   ⍝ Scatter indexing of ω by α. Ranks: 1 ¯
[2]    α←,⍤1 α ⍝ Promote scalar α to vector
[3]    ⎕signal((⍴⍴ω)≡¯1↑⍴α)↓5 ⍝ Length error
[4]    Δ←(⍴ω)⊥⍤1 α-⎕io
[5]    Δ←(⊂,ω)[Δ+⎕io]
     ∇
```

Answers to Exercise 5

```
V+⍤¯1 M
V+⍤0 1 M
V+⍤¯1 ¯1 M
V+⍤0 ¯1 M
V+⍤¯1 1 M

1 1 2⌽V∘.+M
1 2 1⌽M∘.+V

(⍪V)+⍤1 M

V+⍤⊢⍤⊢1 M
```

Answers to Exercise 6

```
V+⍤1 M
V+⍤1 ¯1 M
V+⍤1 1 M

1 2 1⍉V∘.+M or 1 2 2⍉M∘.+V

(<V)+⍤⍤<⍤1 M
```

Answers to Exercise 7

```
     ∇ Δ←join ω
[1]    Δ←( '.'∊⍤⊃>ω) 2⍤⊃↓ω
     ∇
```

```
     ∇ Δ←join2 ω
[1]    Δ←2⍤⊃<>↓ω
     ∇
```

Answers to Exercise 8

```
2 1 2 ⍷(⍵∈'aeiou')+2×⍵∈'bcdfghjklmnpqrstvwxyz'
```

```
    ∇ r←CVC s;C;V
[1]   ⍝ r←boolean vector where 1s indicate start of substrings of s
[2]   ⍝ of the form <Consonant-Vowel-Consonant>
[3]   ⍝ eg, CVC 'the cat in the hat'
[4]   ⍝               ↑           ↑
[5]    V←'aeiou'
[6]    C←'bcdfghjklmnpqrstvwxyz'
[7]    r← 2 1 2 ⍷+⌿> 1 2 ×⍤¨>(<s)∈⍤¨>V⊃C
    ∇
```

Answers to Exercise 9

	$\omega \leftarrow A$	$\omega \leftarrow B$
$<\ddot{\circ}0\ \omega$	B	C
$\supset\ddot{\circ}0\ \omega$	B	B
$<\ddot{\circ}>\ \omega$	B	B
$<\ddot{}>\ \omega$	C	C
$\vdash\ddot{}>\ \omega$	B	B
$\supset\ddot{\circ}>\ \omega$	B	B
$\supset\ddot{}>\ \omega$	C	C

Answers to Exercise 10

```
     ∇ ω←⍺ cddf ω;a;⎕io;i;⎕ec
[1]   ⍝ ω←⍺⌿[⍤¯1]⍵ (compress-dual-disclose on the first axis) (gilhv fn)
[2]    ⎕io←⍬←i←0 ⋄ a←⍴ω←ω⊣⌿[⍤¯1]⍺←⍺⊢⌿[⍤¯1]⍺ ⋄ ⍺←,⍺ ⋄ ω←,ω
[3]   A:→((⍴ω)<i←i+1)⍴B ⋄ ω[i]←<(>⍺[i])⌿[⍤¯1]>ω[i] ⋄ →A
[4]   B:ω←a⍴ω
     ∇
```

Answers to Exercise 11

```
     ∇ ω←⍺ cddl ω;a;⎕io;i;⎕ec
[1]   ⍝ ω←⍺/⍤¯1>ω (compress-dual-disclose on the last axis) (gilhv fn)
[2]    ⎕io←~i←0 ⋄ a←⍴ω←ω⍪⍤¯1>⍺←ω⊢⍤¯1>⍺ ⋄ ⍺←,⍺ ⋄ ω←,ω
[3]   A:→((⍴ω)<i←i+1)⍴B ⋄ ω[i]←<(>⍺[i])/>ω[i] ⋄ →A
[4]   B:ω←a⍴ω
     ∇
```